Efficient Expression Placement through Lazy Code Motion in LLVM

Vinodh Kumar Reddy Kamasani
The Bradley Department of Electrical and Computer Engineering
Virginia Tech
Blacksburg, Virginia, USA
vinodh@vt.edu

Punitha Kamasani
The Bradley Department of Electrical and Computer Engineering
Virginia Tech
Blacksburg, Virginia, USA
punitha@vt.edu

***Abstract*— This project focuses on the problem of computational redundancy in software, where the same calculations are performed multiple times, leading to wasted CPU cycles and reduced performance. The work details the design, implementation, and evaluation of a Lazy Code Motion (LCM) optimization pass within the LLVM compiler infrastructure (version 17.0.6) using its New Pass Manager. LCM is a powerful dataflow analysis technique that aims to eliminate both total and partial redundancies by strategically relocating computations. The primary goal was to develop a functional LCM pass that implements core dataflow analyses (Available Expressions, Anticipated Expressions, Used Expressions, Postponable Expressions), computes optimal placement points for code, performs the necessary IR transformations, and integrates this as a pass plugin. The pass also includes mechanisms for comparative analysis (LCM-E vs LCM-L) and critical edge detection. Evaluation was performed by analyzing the pass's ability to reduce redundancy on various test cases.**

***Keywords*—Lazy Code Motion (LCM), LLVM, Compiler Optimization, Dataflow Analysis, Partial Redundancy Elimination (PRE), Code Motion, Compiler Pass, Intermediate Representation (IR), SSA Form, Dominator Tree, Critical Edge Detection**

I. INTRODUCTION

The high demand for faster and more efficient software places significant emphasis on sophisticated compiler optimizations. Compilers translate high level code to low level machine instructions, aiming for fast execution and minimal resource use. However, programmer written or early stage compiled code often contains inefficiencies, a major one being computational redundancy. This occurs when identical calculations on the same inputs are repeated, wasting CPU cycles and potentially increasing register usage or code size. Identifying and eliminating such redundancies is the key for compiler optimization. Code motion techniques like Lazy Code Motion (LCM) address this by re-arranging code to ensure computations are performed optimally.

The motivation for implementing LCM lies in its ability to eliminate comprehensive redundancy (both total and partial), optimal computation (no new work introduced on any path), and consideration for register usage by placing computations as late as possible (the lazy aspect). LCM offers an effective balance and is an excellent case study for data flow driven optimizations in LLVM. This project's goal was to implement LCM within LLVM's New Pass Manager to explore its practical application and effectiveness.

II. RELATED WORK

Code motion optimizations are fundamental for reducing computational redundancy by relocating expressions within a program's Control Flow Graph (CFG). This includes techniques like Loop Invariant Code Motion (LICM), Common Subexpression Elimination, and Partial Redundancy Elimination (PRE). An expression is considered totally redundant if its value is already computed on all paths to a point p, and partially redundant if computed on same paths to a point p. PRE aims to eliminate both the redundancies by inserting computations on paths where missing. But, early PRE algorithms could sometimes introduce unnecessary computations or increase register pressure.

Lazy Code Motion (LCM), an algorithm developed by Knoop, Ruthing, and Steffen, is a Partial Redundancy Elimination technique based on bidirectional dataflow analysis. Its goal is to achieve optimal computational performance (by eliminating redundant calculations). To do this, LCM analyzes code to determine properties like Available Expressions and Anticipated Expressions. These help define the Earliest and Latest possible positions for computations, ultimately guiding the Insertion points to place calculations as late as possible within valid regions. A known complexity is handling critical edges (source block has multiple successors, destination block has multiple predecessors), which may require edge splitting.

The foundational work on PRE and LCM established core dataflow principles. Modern advancements, especially with SSA-based IR like LLVM's, have led to SSA-based PRE algorithms (e.g., Chow et al.) that can be more efficient. Production compilers like Clang/LLVM often integrate redundancy elimination within broader frameworks like Global Value Numbering (GVN), which identifies redundancy based on value equivalence. Implementing such passes in frameworks like LLVM requires careful handling of its pass manager, analysis dependencies, and IR modification rules. While classic LCM is effective, current art often involves integrated, SSA-based techniques, though understanding LCM provides valuable insight.

III. PROPOSED METHOD

This section details the design and implementation of the Lazy Code Motion (LCM) pass within LLVM 17.0.6.

A. Environment Setup

We developed it on on Ubuntu 22.04.5 LTS, using LLVM/Clang 17.0.6, CMake 3.22.1, g++ 11.4.0 (C++17), targeting x86_64-pc-linux-gnu. The implementation uses LLVM C++ APIs and integrates with the New Pass Manager (NPM) as a dynamically loaded plugin.

B. Core Dataflow Framework (Dataflow class)

A generic Dataflow class was implemented to provide a reusable engine for LCM's analyses. It uses an iterative worklist algorithm and is configurable for direction (FORWARD / BACKWARD), uses llvm::BitVector for dataflow facts, allows specifying boundary/initial values, a meet operator function, and a transfer function. It stores IN/OUT sets for each basic block.

C. Base Analysis Pass (AnalysisPassBase)

An AnalysisPassBase class was created for common functionality shared by specific dataflow analysis passes. It handles expression domain management (mapping expressions to BitVector indices), stores GEN/KILL sets, holds a Dataflow instance, provides shared printing utilities, and defines an abstract calculateGenKillSets method for derived classes.

D. Dataflow Analyses Implementation

Four core dataflow analyses inheriting from AnalysisPassBase were implemented:

- **Available Expressions (AvailableExpressions):** Determines expressions computed along all paths to a block's entry. It's a forward analysis with intersection as the meet operator. GEN[B] are expressions computed in B and not killed in B; KILL[B] are expressions whose operands are redefined in B.

  $$OUT[B] = (IN[B] - KILL[B]) \cup GEN[B]$$

- **Anticipated Expressions (AnticipatedExpressions):** Determines expressions computed along all paths from a block's exit before operands are redefined (aka Very Busy Expressions). It's a backward analysis with intersection. GEN/KILL sets are computed by iterating backward through a block's instructions.

  $$IN[B] = (OUT[B] - KILL[B]) \cup GEN[B]$$

- **Used Expressions (UsedExpressions):** Determines expressions computed in a block B whose results might be used on a path starting from B's exit. It's a backward analysis with union as the meet operator. GEN[B] are expressions whose result is used as an operand in B and the expression isn't killed later in B (backward); KILL[B] are expressions computed in B.

  $$IN[B] = (OUT[B] - KILL[B]) \cup GEN[B]$$

- **Postponable Expressions (PostponableExpressions):** Determines how far an expression's computation can be delayed from its earliest placement without causing recomputation. It's a forward analysis; correction noted in source with intersection (correction noted in source). Its transfer function depends on the results of UsedExpressions. These analyses provide the dataflow information for the main LCM pass.

  $$OUT[B] = (IN[B] \cup GEN[B]) - KILL[B]$$

E. Lazy Code Motion Pass Implementation (LazyCodeMotion)

The LazyCodeMotion class is an LLVM FunctionPass using New Pass Manager.

Prerequisite Analyses Usage: Retrieves results from Available Expressions, Anticipated Expressions, Used Expressions, and standard DominatorTreeAnalysis using the FunctionAnalysisManager.

Calculation of LCM Placement Sets:

EARLIEST[B]: ANTIC_IN[B] & (~AVAIL_IN[B] | USED_IN[B])

LATEST_IN[B]: Iteratively computed, identifies blocks up to which placement is possible without unnecessary computations.

INSERT[B]:LATEST_IN[B]&(EARLIEST[B]|~LATEST_IN_Preds)

This aims for the latest possible placement.

Phase 1: Temporary Insertion Logic: Iterates blocks and expressions in INSERT[B] (LCM-L mode) or EARLIEST[B] (LCM-E mode). A flag useEarliestInsertion controls this. New temporaries (lcm.tmp = ...) are inserted after PHI/debug instructions if operands dominate the insertion point (checked using DominatorTreeAnalysis). Inserted temporaries are tracked.

Phase 1.5: Critical Edge Detection Logic: Diagnostically checks if optimal placement might have involved a critical edge by examining blocks B with multiple predecessors where an expression e is ANTIC_IN[B] but not AVAIL_IN[B], and a predecessor P also has multiple successors. It prints info but doesn't split edges.

Phase 2: Replacement Map Construction: For each original binary operator, it finds the highest dominating inserted temporary computing the same expression and maps the original instruction to this temporary in replacementMap.

Phase 3: Replacement and Deletion Logic: Iteratively replaces operands of instructions with their corresponding temporaries from replacementMap if dominance checks pass. Original instructions in replacementMap that are now dead code are erased.

F. LLVM Pass Plugin Integration

The custom analyses and LazyCodeMotion pass are packaged as a shared object (UnifiedPass.so). Integration uses llvmGetPassPluginInfo() to register analyses with FunctionAnalysisManager via PassBuilder :: registerAnalysisRegistrationCallback and the "lcm" pipeline (and print pipelines) via PassBuilder :: registerPipelineParsingCallback. This allows invocation using opt-17.

IV. Evaluation Plan

A. Evaluation Metrics

The Primary metrics used for the evaluation are:

- **Redundancy Elimination:** Assessed by manual inspection and comparison of LLVM IR before and after the LCM pass, looking for removal of redundant computations and correct placement of temporaries.
- **Code Placement Analysis:** Examining where the LCM pass (in LCM-E and LCM-L modes) places computations.
- **Critical Edge Detection:** Verifying the diagnostic output of the critical edge detection logic.

B. Benchmark Programs (Test Cases)

Several C code examples were used to test different aspects of the LCM implementation Major test cases include, test.c (A general CFG with branches to test basic code motion.) ,test_partial_redundancy.c ,test_critical_edge_trigger.c ,test_complex_cfg.c, test_loop_invariant.c

C. Experimental Setup

Test Environment: Ubuntu 22.04.5 LTS, LLVM/Clang 17.0.6, CMake 3.22.1, g++ 11.4.0.

- **Configurations Compared for each test case:** The performance and behavior of the custom Lazy Code Motion (LCM) pass were evaluated against the following configurations for each test case:
  - Baseline (Unoptimized): clang-17 -O0 then opt-17 -passes=mem2reg (SSA form).
  - Custom LCM Pass (Latest Mode, LCM-L): Baseline + custom LCM pass (-passes=lcm, useEarliestInsertion=false).
  - Custom LCM Pass (Earliest Mode, LCM-E): Baseline + custom LCM pass (-passes=lcm, useEarliestInsertion=true).
  - Standard LLVM Optimization: clang-17 -O1 as a visual benchmark.
- **Methodology:** The evaluation was conducted using the following steps for each benchmark program:
  - C source compiled to LLVM bitcode (-O0, -fno-discard-value-names,-Xclang-disable-O0 optnone).
  - mem2reg pass for SSA conversion.
  - Custom LCM pass plugin loaded into opt-17 (-load-pass-plugin=./build/UnifiedPass.so -passes=lcm).
  - Output human-readable LLVM assembly (.ll files) using -S.
  - Comparison using diff -u and manual inspection of .ll files; analysis of diagnostic logs.
- **Critical Edge Detection:** Verified the diagnostic output of the critical edge detection logic in the terminal.
  - Used the test cases designed to contain critical edges (e.g., test_critical_edge_trigger.c) as well as test cases without them (e.g., test.c)
  - The diagnostic output printed to the terminal to confirm that the logic correctly identified and reported potential critical edge insertion scenarios when they existed

V. Discussion

A. Effectiveness of Implemented LCM:

The primary goal of eliminating computational redundancy was achieved for successful test cases (test.c, test_partial_redundancy.c, test_critical_edge_trigger.c), where a + b was correctly hoisted. This confirmed the core algorithm's functionality.

```
; Function Attrs: noinline nounwind uwtable
define dso_local i32 @bar(i32 noundef %a, i32 noundef %b) #0 {
entry:
  %add = add nsw i32 %a, %b
  %cmp = icmp sgt i32 %a, 10
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  %add1 = add nsw i32 %a, %b
  %sub = sub nsw i32 %a, %b
  br label %if.end

if.else:                                          ; preds = %entry
  %mul = mul nsw i32 %a, %b
  br label %if.end

if.end:                                           ; preds = %if.else, %if.then
  %x.0 = phi i32 [ %sub, %if.then ], [ %add, %if.else ]
  %y.0 = phi i32 [ %add1, %if.then ], [ %mul, %if.else ]
  %add2 = add nsw i32 %a, %b
  %mul3 = mul nsw i32 %x.0, %y.0
  ret i32 %mul3
}
```

Figure 1. LLVM IR for the Baseline

```
; Function Attrs: noinline nounwind uwtable
define dso_local i32 @bar(i32 noundef %a, i32 noundef %b) #0 {
entry:
  %lcm.tmp = add i32 %a, %b
  %lcm.tmp1 = sub i32 %a, %b
  %lcm.tmp2 = mul i32 %a, %b
  %cmp = icmp sgt i32 %a, 10
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  %lcm.tmp3 = sub i32 %a, %b
  br label %if.end

if.else:                                          ; preds = %entry
  %lcm.tmp4 = mul i32 %a, %b
  br label %if.end

if.end:                                           ; preds = %if.else, %if.then
  %x.0 = phi i32 [ %lcm.tmp1, %if.then ], [ %lcm.tmp, %if.else ]
  %y.0 = phi i32 [ %lcm.tmp, %if.then ], [ %lcm.tmp2, %if.else ]
  %lcm.tmp5 = mul i32 %x.0, %y.0
  ret i32 %lcm.tmp5
}
```

Figure 2. LLVM IR after applying the LCM Pass

B. Comparison of LCM-E vs LCM-L:

The switch between LCM-Earliest (LCM-E) and LCM-Latest (LCM-L) showed differing placements in test.c, where LCM-E placed some computations later while LCM-L hoisted them higher. This illustrates the trade-off: LCM-E minimizes movement, LCM-L delays computation. In other cases (test_partial_redundancy.c, test_critical_edge_trigger.c), both modes produced similar final code because the replacement logic favored the highest dominating temporary, rendering some LCM-E insertions dead.

```
; Function Attrs: noinline nounwind uwtable
define dso_local i32 @bar(i32 noundef %a, i32 noundef %b) #0 {
entry:
  %lcm.tmp = add i32 %a, %b
  %lcm.tmp1 = sub i32 %a, %b
  %lcm.tmp2 = mul i32 %a, %b
  %cmp = icmp sgt i32 %a, 10
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  %lcm.tmp3 = sub i32 %a, %b
  br label %if.end

if.else:                                          ; preds = %entry
  %lcm.tmp4 = mul i32 %a, %b
  br label %if.end
```

Figure 3. LLVM IR for LCM-L (Default)

```
; Function Attrs: noinline nounwind uwtable
define dso_local i32 @bar(i32 noundef %a, i32 noundef %b) #0 {
entry:
  %lcm.tmp = add i32 %a, %b
  %cmp = icmp sgt i32 %a, 10
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  %lcm.tmp1 = sub i32 %a, %b
  br label %if.end

if.else:                                          ; preds = %entry
  %lcm.tmp2 = mul i32 %a, %b
  br label %if.end
```

Figure 4. LLVM IR for LCM-E

C. *Comparison with Standard LLVM Optimization:*

LLVM's -O1 also eliminated primary redundancies but performed significantly more transformations (e.g., DCE, CFG simplification with select in test_partial_redundancy.c). This highlights that the custom LCM pass is one specific technique, while standard optimization levels integrate many synergistic passes for holistic improvement.

D. *Critical Edge Handling:*

The implemented logic successfully detected potential critical edge insertion scenarios (e.g., in test_critical_edge_trigger.c) and correctly reported no issues where none existed (e.g., test.c). However, it only reports these situations and does not perform edge splitting or code duplication, meaning placement might remain suboptimal in such cases.

```
  Checking Block: if.then
    Block if.then has multiple predecessors? No
  Finished Checking Block: if.then
  Checking Block: if.else
    Block if.else has multiple predecessors? No
  Finished Checking Block: if.else
  Checking Block: if.end
    Block if.end has multiple predecessors? Yes
    AVAIL_IN : c * 10
    ANTIC_IN : a + b
      Checking Expr 0 [c * 10]: ANTIC=0, AVAIL=1
      Checking Expr 1 [a + b]: ANTIC=1, AVAIL=0
        Heuristic PASSED for Expr 1 [a + b]
        Checking Predecessors:
          Pred P: if.else
            Pred P has multiple successors? No
          Pred P: if.then
            Pred P has multiple successors? No
      Checking Expr 2 [y.0 + add1]: ANTIC=0, AVAIL=0
  Finished Checking Block: if.end
LCM: Phase 1.5 - Finished Checking.
```

Figure 5. Diagnostic output from Critical Edge Detection

E. *Limitations and Known Issues:*

- Runtime Crash: Consistent crash with test_loop_invariant.c due to an LLVM BitVector::resize issue, preventing analysis of loop invariant motion with that test case.
- Incomplete Redundancy Elimination: Some potential redundancies were missed in certain CFG patterns, as noted earlier.
- Dead Temporaries: The insertion phase often generates temporary variables that become dead code after replacement, relying on a subsequent DCE pass for removal.
- No Critical Edge Transformation: Only detection, not resolution.

Overall, the pass demonstrated LCM principles, E/L modes, and critical edge detection, while stating that there are still areas for improvement

VI. BENCHMARKS

Comparing unoptimized code with LCM-E, LCM-L (which serves as default full LCM as per Readme on how to run.txt). The "Original Redundant Expressions" column represents the count before any optimization (i.e., the number of expressions our LCM pass targets).

Table 1. Benchmark Results of LCM

Benchmark	Opt Mode	Redundant Expressions		Temporaries Inserted
		Before Opt	Eliminated	
test.c Partial Redundancy + Branch Merge	LCM-E	6	0	0
	LCM-L		6	6
test2.c Redundancy Across Branches	LCM-E	6	6	7
	LCM-L		6	7
test3.c Redundancy Inside Loop	LCM-E	4	4	7
	LCM-L		3	3
test4.c Nested Branch Redundancy	LCM-E	6	6	9
	LCM-L		6	11
test5.c Multi-path Partial Redundancy	LCM-E	8	8	12
	LCM-L		8	16
test_complex_cfg.c	LCM-E	13	13	8
	LCM-L		11	7
test_partial_ redundancy.c	LCM-E	3	3	5
	LCM-L		3	5

More temporaries are inserted by the Lazy Code Motion (LCM) optimization because it's trying to avoid re-doing the same calculations. Depending on how complex code's branching and looping structures are (like in test_complex_cfg.c or test5.c), and the specific strategy the optimizer uses (like placing calculations as early or as late as possible), it might not find a single, perfect spot to store a reused calculation.

VII. Impact of code on the Optimization technique

Based on the observed results from the test cases, here's how different types of code involving loops performed under unoptimized, LCM-L, and LCM-E executions:

A. *Loops with Internal Redundancy and Invariants:*

For loops containing expressions that are both internally redundant and loop-invariant (as exemplified by test3.c, also referred to as loop_test), both LCM-L and LCM-E executions demonstrated significant benefits over unoptimized code.

- Unoptimized: In test3.c, the expression a + b was computed in the entry block before the loop and then redundantly recomputed inside the loop body during each iteration.
- LCM-L Execution: This mode successfully hoisted the loop-invariant computation a + b to the entry block. Original computations of a + b within and after the loop were replaced by this hoisted temporary. This resulted in 3 original instructions being deleted and 3 temporaries being inserted.
- LCM-E Execution: This mode also hoisted a + b to the entry block. However, it made more distributed temporary insertions related to loop-carried values and the loop control structure itself (e.g., in for.cond, for.body, for.inc, for.end). This led to the deletion of 4 original instructions and the insertion of 7 temporaries.

B. *General Observations on Loop-Related Code:*

For general code structures that involve loops and partial redundancies, such as test_complex_cfg.c which had (a + b) * 3 inside a branch within a larger conditional structure that itself could be part of a loop if the function was called repeatedly, both LCM-L and LCM-E were effective. They hoisted common parts like a+b to the entry of the function. In these more complex, non-loop-focused but potentially loop-invoked scenarios, LCM-E tended to eliminate slightly more original instructions and insert slightly more temporaries than LCM-L, placing them at the earliest valid points within the control flow.

C. *Pure Loop-Invariant Code:*

The test_loop_invariant.c case was designed to specifically test the hoisting of purely loop-invariant code (an expression like a + b computed repeatedly inside a loop with no other complex interactions).

- Unoptimized: The invariant expression a + b would be re-calculated in every iteration of the loop.
- LCM-L & LCM-E Execution: Direct observation of the optimized .ll files for this specific scenario was not possible. The execution logs consistently showed that the LCM pass (in both L and E modes) crashed with an assertion failure (llvm::SmallVectorTemplateCommon::back() !empty()) during the AvailableExpressions analysis phase when processing test_loop_invariant.mem2reg.bc.

VIII. Conclusion and Future Work

A. *Effectiveness of Implemented LCM:*

This project successfully implemented a Lazy Code Motion (LCM) optimization pass in LLVM 17 using the New Pass Manager. It involved developing a dataflow framework and the necessary analyses (Available, Anticipated, Used, Postponable Expressions). The transformation included calculation of EARLIEST, LATEST_IN, INSERT sets, temporary insertion with dominance checks, and replacement/deletion of redundant instructions. Enhancements like LCM-E/L modes and critical edge detection were incorporated. Experiments showed effectiveness in eliminating redundancies in several test cases, and the E/L comparison highlighted placement strategy differences. Critical edge detection worked as intended.

B. *Future Work:*

- Resolve Runtime Crash: Debug and fix the BitVector assertion failure with test cases that have loop structure like test_loop_invariant.c.
- Implement Critical Edge Handling: Move beyond detection to actual CFG modification using utilities like SplitCriticalEdge, updating PHI nodes and dominator info.
- Enhance Redundancy Elimination: Explore techniques for more complex partial redundancies.
- Expand Expression Scope: Handle more expression types (comparisons, GEPs, safe function calls).
- Performance Benchmarking: Conduct rigorous performance evaluation with larger benchmarks (e.g., SPEC) and measure execution time differences to quantify real-world impact and register pressure trade-offs.
- LLVM Migration: Ensure compatibility with newer Ubuntu and LLVM versions

References

[1] J. Knoop, O. Rüthing, and B. Steffen, "Lazy Code Motion," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, New York, NY, USA: ACM, 1992, pp. 224–234, doi: 10.1145/143095.143114.

[2] J. Knoop, O. Rüthing, and B. Steffen, "Optimal Code Motion: Theory and Practice," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 4, pp. 1117–1155, Jul. 1994, doi: 10.1145/183432.183436.

[3] E. Morel and C. Renvoise, "Global Optimization by Suppression of Partial Redundancies," *Commun. ACM*, vol. 22, no. 2, pp. 96–103, Feb. 1979, doi: 10.1145/359060.359063.

[4] K.-H. Drechsler and M. P. Stadel, "A variation of Knoop, Rüthing and Steffen's LAZY CODE MOTION," *ACM SIGPLAN Notices*, vol. 28, no. 5, pp. 29–38, May 1993, doi: 10.1145/155183.155186.

[5] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, F. Catthoor, and H. Corporaal, "Integrating Advanced Code Motion Transformations in a Realistic Compiler," *Softw. Pract. Exp.*, vol. 33, no. 15, pp. 1417–1443, Dec. 2003, doi: 10.1002/spe.543.

[6] F. Chow et al., "A New Algorithm for Partial Redundancy Elimination Based on SSA Form," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, New York, NY, USA: ACM, 1997, pp. 273–286, doi: 10.1145/258915.258992.

[7] LLVM Project, "Writing an LLVM Pass," *LLVM Documentation*. [Online]. Available: https://llvm.org/docs/WritingAnLLVMPass.html